Hashing in Quantum Decision Diagrams

1 Motivation & Background

Quantum circuit simulation is useful for testing and comparing quantum algorithms before running them on real, expensive quantum hardware. A direct state-vector simulator stores all 2^n complex amplitudes of an n-qubit state, so memory grows exponentially with the number of qubits. This makes alternative representations important for studying larger or more structured circuits.

Quantum Multiple-valued Decision Diagrams (QMDDs) reduce part of this cost by representing states and operations as weighted directed graphs instead of flat vectors or matrices. They recursively decompose the represented object and store repeated substructures only once. This sharing can make QMDDs compact for structured circuits, but it also creates a lookup problem: before adding a new node, the simulator must check whether an equivalent node already exists in the unique table.

2 Problem & Research Questions

This project studies node-table behaviour in QolDDer-QMDD, a QMDD-based quantum circuit simulator. In the original implementation, candidate nodes are found using a linked-list-style lookup structure, which may require scanning many stored nodes. Hash tables can reduce this search cost by sending each candidate node to a bucket, but their performance depends on how well the hash function distributes the QMDD node keys. If many different nodes end up in the same bucket, lookup can still require many comparisons. Therefore, the project studies how Shor-generated QMDD nodes should be organised during lookup.

The research question is: how does the structure of Shor's Algorithm influence node-table calls and hashing behaviour in QMDD-based simulation, and how can this behaviour be explained from the resulting QMDD representation to inform hash-table design?

3 Method & Experiments

QolDDer-QMDD was instrumented with counters for unique-table activity. The recorded metrics include lookups, insertions, candidate checks, label equality checks, collisions, maximum bucket length, stored entries, and load factor. Three unique-table configurations were compared: the *original linked-list-style baseline*, a *basic hash table* based mainly on child-node information, and a *label-aware hash table* that also includes high-edge-label information in bucket selection. Deutsch-Jozsa circuits were used as validation benchmarks, while Shor-style circuits were used as the main workloads. The main benchmark was the gate-level Shor instance (N=15, a=2), with (N=21, a=2) used as a larger validation case.

4 Results & Insights

For the main Shor benchmark, all three configurations performed the same number of lookups and insertions and stored the same number of entries, so the simulated QMDD workload stayed unchanged. The difference was in lookup cost.

The linked-list baseline required 3.45 million candidate checks, while the basic hash table reduced this to 35,248. However, the basic hash still produced a bucket of length 115. The label-aware hash reduced candidate checks further to 6,833 and reduced the maximum bucket length to 7. The larger (N=21, a=2) benchmark showed the same trend, suggesting that the effect is not limited to the smallest Shor instance. This shows that hashing helps, but also that the hash function matters: Shor-generated nodes often have similar child-node structure while differing in edge labels, so using label information earlier gives a better bucket distribution.

Unique-table mode	Lookups	Entries	Candidate checks	Label checks	Collisions	Max. bucket
Linked-list baseline	6,689	2,031	3,448,658	15,712	N/A	N/A
Basic hash	6,689	2,031	35,248	25,424	1,323	115
Label-aware hash	6,689	2,031	6,833	4,671	885	7

Table 1: Main results for the Shor (N=15, a=2) benchmark. All three unique-table configurations perform the same number of lookups and store the same number of entries, so the simulated QMDD workload is unchanged. Compared with the basic hash table, the label-aware hash reduces candidate checks by 80.6%, reduces label equality checks by 81.6%, and lowers the maximum bucket length from 115 to 7.

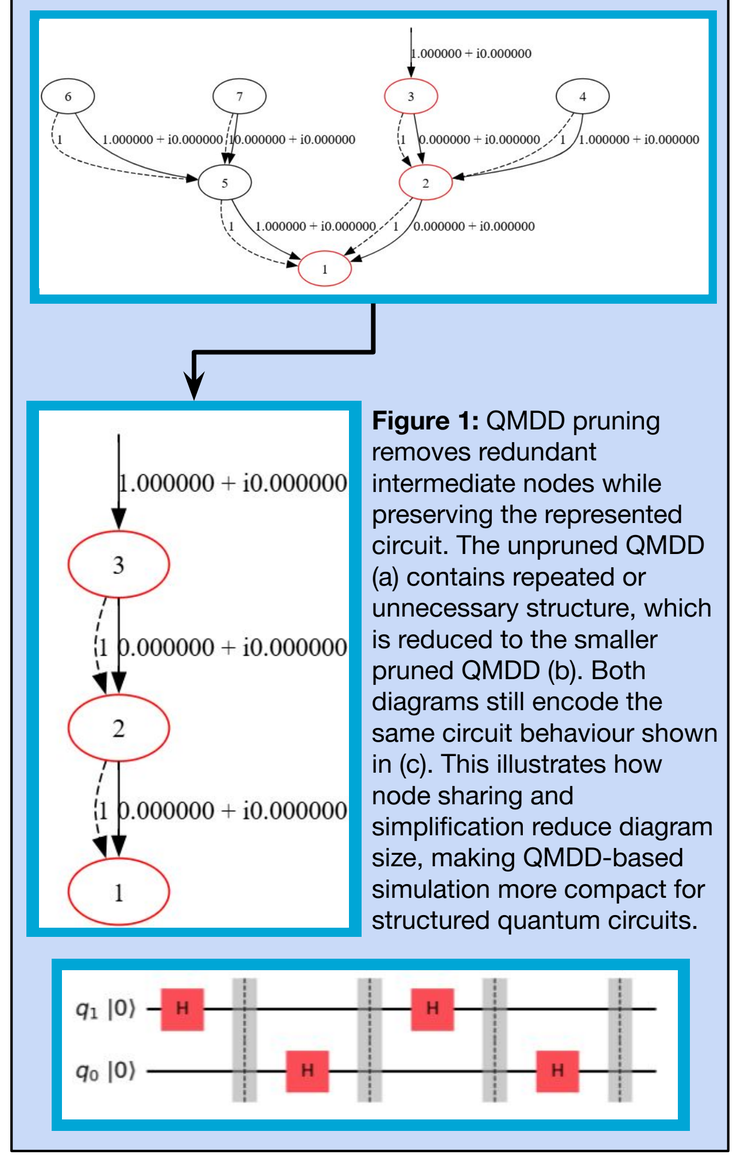

Figure 1: QMDD pruning removes redundant intermediate nodes while preserving the represented circuit. The unpruned QMDD (a) contains repeated or unnecessary structure, which is reduced to the smaller pruned QMDD (b). Both diagrams still encode the same circuit behaviour shown in (c). This illustrates how node sharing and simplification reduce diagram size, making QMDD-based simulation more compact for structured quantum circuits.

TU Delft

Caspar Backx · CSE3000 Research Project
TU Delft EEMCS · May 2026
Supervisors: Tim Coopmans, Juul Sanders